# Oracle10g Advanced SQL Programming

Student Workbook

itcourseware

Training Materials For IT Professionals

*Oracle10g Advanced SQL Programming*

Brian Peasland, Robert Roselius, and Robert Seitz

Published by ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, CO 80112

**Editor:** Jan Waleri

**Editorial Assistant:** Dana Howell

**Special thanks to:** Many instructors whose ideas and careful review have contributed to the quality of this workbook and the many students who have offered comments, suggestions, criticisms, and insights.

# Contents

# Chapter 1 - Course Introduction

## COURSE OBJECTIVES

❈ Apply the basic theory behind relational database design.

❈ Contribute to all phases of database design and development.

❈ Use all aspects of subqueries.

❈ Apply Oracle's features for querying hierarchical data models.

❈ Use Oracle's Object-Relational Model.

❈ Create object types.

❈ Use Oracle's collection types in SQL.

❈ Select appropriate date-related datatypes for your applications.

❈ Use Oracle's regular expression SQL functions to perform pattern matching and string manipulation.

❈ Create and manage temporary tables.

❈ Establish goals in SQL tuning to improve performance.

❈ Use Oracle Database 10*g*'s tuning tools.

❈ Describe how indexes are used in RDBMSs, and use them effectively.

❈ Use the various analytic functions provided by Oracle to perform sophisticated analysis.

❈ Use SQL*Plus to format reports and extract data.

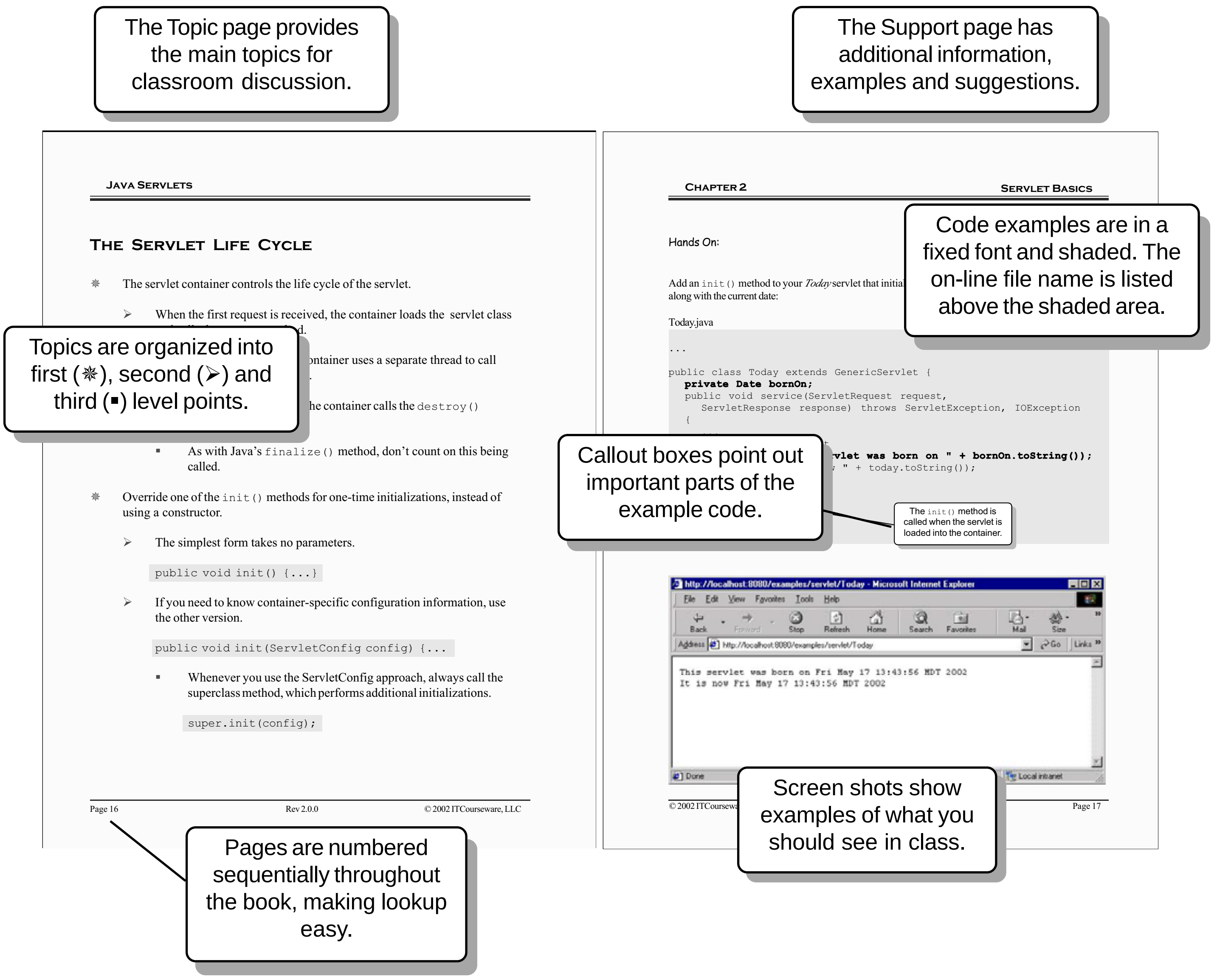
## COURSE OVERVIEW

- ❅ **Audience**: Application developers, database administrators, system administrators, and users who write applications and procedures that access an Oracle10*g* database.
- ❅ **Prerequisites**: *Oracle10g SQL Programming*
- ❅ **Classroom Environment:**
  - ➢ A workstation per student.
  - ➢ Oracle Database 10*g*, with Oracle's Sample Schemas installed.

## USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. Hands On sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.



The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.

JAVA SERVLETS

THE SERVLET LIFE CYCLE

❋ The servlet container controls the life cycle of the servlet.

➢ When the first request is received, the container loads the servlet class

d.

ontainer uses a separate thread to call

.

he container calls the `destroy()`

▪ As with Java's `finalize()` method, don't count on this being called.

❋ Override one of the `init()` methods for one-time initializations, instead of using a constructor.

➢ The simplest form takes no parameters.

```
public void init() {...}
```

➢ If you need to know container-specific configuration information, use the other version.

```
public void init(ServletConfig config) {...
```

▪ Whenever you use the ServletConfig approach, always call the superclass method, which performs additional initializations.

```
super.init(config);
```

Page 16 Rev 2.0.0 © 2002 ITCourseware, LLC

Topics are organized into first (❋), second (➢) and third (▪) level points.

Pages are numbered sequentially throughout the book, making lookup easy.

CHAPTER 2 SERVLET BASICS

Hands On:

Add an `init()` method to your *Today* servlet that initial along with the current date:

Today.java

```
...

public class Today extends GenericServlet {
   private Date bornOn;
   public void service(ServletRequest request,
      ServletResponse response) throws ServletException, IOException
   {
```

```
vlet was born on " + bornOn.toString());
 " + today.toString());
```

The `init()` method is called when the servlet is loaded into the container.

Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

http://localhost:8080/examples/servlet/Today - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Mail Size
Address http://localhost:8080/examples/servlet/Today
Go Links
Done
Local intranet

```
This servlet was born on Fri May 17 13:43:56 MDT 2002
It is now Fri May 17 13:43:56 MDT 2002
```

© 2002 ITCoursewa Page 17

Screen shots show examples of what you should see in class.

## SUGGESTED REFERENCES

Celko, Joe. 1999. ***Joe Celko's SQL for Smarties: Advanced SQL Programming, Second Edition***. Academic Press/ Morgan Kaufman, San Francisco, CA. ISBN 1558605762

Celko, Joe. 1997. ***Joe Celko's SQL Puzzles and Answers***. Morgan Kaufmann, San Francisco,CA. ISBN 1558604537.

Date, C.J. and Hugh Darwen. 1996. ***A Guide to The SQL Standard, Fourth Edition***. Addison-Wesley, Reading, MA. ISBN 0201964260.

Freeman, Robert G. 2004. ***Oracle Database 10g New Features***. McGraw-Hill Osborne Media, Emeryville, CA. ISBN 0072229470.

Gennick, Jonathan. 2004. ***Oracle Sql*Plus Pocket Reference, Third Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596008856.

Gennick, Jonathan. 2004. ***Oracle SQL*Plus : The Definitive Guide, Second Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596007469.

Gruber, Martin. 2000. ***SQL Instant Reference, Second Edition***. SYBEX, Alameda, CA. ISBN 0782125395.

Kline, Kevin. 2004. ***SQL in a Nutshell, Second Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596004818.

Kreines, David. 2003. ***Oracle Data Dictionary Pocket Reference***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596005172.

Loney, Kevin. 2004. ***Oracle Database 10g: The Complete Reference***. McGraw-Hill/Osborne Media, Emeryville, CA. ISBN 0072253517.

Mishra, Sanjay. 2004. ***Mastering Oracle SQL, Second Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596006322.

***www.dbasupport.com***
***www.hot-oracle.com***
***www.oracle.com***
***www.quest-pipelines.com***
***www.searchdatabase.com***
***http://tahiti.oracle.com/***

Your single most important reference is the SQL Reference book, which is part of the Oracle Database Online Documentation. You may have received this on CD-ROM with your Oracle distribution. If not, you can access it online at Oracle's website. This is the official, complete description of Oracle's implementation of SQL. It includes many examples and discussions.

An easy way to find it is to go to:

***http://tahiti.oracle.com/***

Find the documentation for your version of Oracle. Locate the SQL Reference and open the HTML table of contents.

If you have web access in the classroom, open a browser now and find the SQL Reference. Set a browser bookmark and have the SQL Reference at hand throughout this class.

# CHAPTER 2 - DATABASE DESIGN CONCEPTS

## OBJECTIVES

- ❈ Apply the basic theory behind relational database design.
- ❈ Determine the data model's entities and their attributes.
- ❈ Normalize tables in a relational database design.
- ❈ Categorize the operations a database system performs on data.
- ❈ Contribute to all phases of database design and development.

## RELATIONAL DATABASES

- A Relational Database accepts and presents data according to rules based on the Relational Model.
- A Relational Database Management System (RDBMS) is the software that implements this capability.
- The Relational Model describes a particular way of representing data and constructing expressions to operate upon it.
  - Alternative models include the Hierarchical and Network models.
  - The Relational Model, first published by Dr. E. F. Codd in 1970, bases itself on set theory and has several goals:
    - Description of data independent of machine representation.
    - Independence of applications and users from data representation.
    - Provision of a high-level data language, based on predicate calculus.
    - Ability to interact with data independent of ordering, indexing, and access paths.
- In 1985, Dr. Codd published what are now known as Codd's Twelve Rules, describing the minimum requirements for a system to qualify as an RDBMS.
  - The Relational Model remains the predominant model for large databases.
  - Very few RDBMS products faithfully implement all aspects of the Relational Model.
  - Oracle, like most RDBMS products, provides for a practical subset of Codd's Rules and formal relational algebra.

Codd's Twelve (really, thirteen) Rules:

1. All information is represented as column values in rows of tables.

2. Each item of information is accessible by identifying a table, a column in that table, and the primary key of the row that contains the item.

3. The intersection of a specific row and column might have missing or inapplicable information (a "null value").

4. The entire definition of the database is available in a set of tables, which are a database in their own right (a System Catalog or Data Dictionary).

5. The system provides a single, comprehensive language for defining the structure of the database (Data Definition Language (DDL)) and manipulating and querying the data in it (Data Manipulation Language (DML)).

6. A view (virtual table) that is theoretically updatable must be actually updatable.

7. Users can identify a set of rows for insertion, updating, or deletion, as well as for retrieval.

8. Users and applications are not affected by changes in the underlying physical storage format of the language.

9. Users and applications are not affected by changes in the underlying logical storage of the data that do not actually remove information.

10. Integrity constraints are enforced by the system, not by users or applications.

11. Users and applications are not affected by changes in the physical distribution of storage of the data.

12. The system provides no means of subverting the rules and constraints of the system.

And Rule 0:
0. The system maintains itself through its relational abilities.

## THE RELATIONAL MODEL

- The Relational Model applies specific mathematical techniques (relational algebra) and logic (predicate calculus) to the problem of shared access to large amounts of computer data.

- The Relational Model is the source of much of the terminology underlying SQL and SQL databases.

  - The fundamental element is the *domain* — a datatype (**NUMBER**: the domain consisting of all possible number values representable as an Oracle **NUMBER**).

    - An *attribute* consists of an attribute name and its type (its domain).

    - An *attribute value* consists of an attribute and a value of that attribute's type.

  - A *tuple* is a set of attribute values.

  - A *relation* consists of:

    - Its heading — a set of attributes (their names and types).
    - Its body — a specific set of tuples, each consisting of attribute values.

Relations, written in mathematical form:

Company(Warehouse)
Warehouse(WarehouseID, Name, Description, LocationID)
Customer(CustomerID, Name, Street, City, State, Country, PostalCode, Email)
PhoneNumber(CustomerID, PhoneType, Number)

To get an idea of how the mathematical aspects of the Relational Model ... er, relate ... to SQL databases, we can observe a loose correspondence between components of each:

| Relational Model | SQL Database |
|---|---|
| Domain | Datatype |
| Attribute | A column definition |
| Relation | A table or result set |
| Tuple | A row in a table or result set |
| Unary relation | Table or result set with one column (eg., the **DUAL** table) |
| Binary relation | Table or result set with two columns |
| Ternary relation | Table or result set with three columns |
| Arity | Number of columns in table or result set |
| Cardinality | Number of rows in a table or result set |
| Selection | **... WHERE ...** |
| Projection | **SELECT DISTINCT ...** |
| Cartesian product | **SELECT ... FROM a, b;** |
| Set union | **SELECT ... UNION SELECT ...** |
| Set difference | **SELECT ... MINUS SELECT ...** |
| Candidate key | Any of the primary or unique keys of a table |

Note:
This is NOT intended to be a rigorous comparison of the Relational Model with SQL databases!

## RELATIONAL OPERATIONS

- Relational Algebra defines a few basic operations for a relation, as well as several operations built on the basic ones.
  - Selection — The set of tuples in a relation, for which a logical proposition is true.
  - Projection — The set of tuples consisting of the values for some of the attributes in a relation.
  - Cartesian Product — The set of all combinations of tuples from two relations.
  - Set Union — All tuples belonging to either, or both, of two relations.
  - Set Difference — All tuples belonging to one relation, which are not also in the second of two relations.

The degree to which an RDBMS supports true relational algebra can be debated. All would agree that Oracle, like nearly every other RDBMS, does not fully support the Relational Model. The same is true for SQL itself. This does not necessarily detract from the practicality and power of the RDBMS or of SQL. However, we often refer to RDBMSs as "SQL databases" rather than "relational databases" as a reminder of their differences.

Some differences between the formal Relational Model (RM) and real-world RDBMSs include:

- The RM does not allow **NULL** values.
- Tuples are unordered; in SQL, column order is often significant (**SELECT ***, **INSERT**, etc.)
- Duplicate tuples are not allowed in a relation; duplicates can occur in SQL (in non-**DISTINCT** result sets, in tables with no unique or primary key, in a **UNION ALL**, etc.)
- In the RM, every attribute must be named; SQL allows unnamed attributes (**SELECT 2+2**, ...)

There are others, but you get the idea. It is not necessary to study relational algebra and set theory in order to advance your SQL and database design skills. However, some familiarity with the theoretical underpinnings of the system and language we work with can provide helpful insights.

It is natural to suppose that the term "Relational Database" refers to the relationships (foriegn keys, joins) between the tables in the database. Actually, the term refers to the relational model, in which the "relation" is the set of attributes — a table.

## THE DATABASE DESIGN PROCESS

- One of the earliest phases of database design is development of the *Conceptual Model*.
    - End users, customers, functional and process experts, managers, and subject-matter experts (both in the subject business, and in any existing legacy systems) are among those who can help the database designer during this process.
    - During this phase, the database designer helps identify entities, their attributes, and dependencies and associations between them.
        - The cardinality (one-to-one, one-to-many, many-to-many) of associations also become apparent.
    - Entity-Relationship diagrams help document this phase.
- Development of the *Logical Model* from the Conceptual Model provides concrete mapping of conceptual elements into specific database structures (tables, columns, constraints, etc.)
    - Normalization of tables typically occurs during this phase.
- Finally, a *Physical Model* specifies how the database will be stored on a computer.
    - Tablespaces, and their datafiles and storage parameters.
    - Physical requirements of the database host system.
    - Indexes, instance parameters, access methods, etc.

There are many variations on the process of database design, of varying degrees of formality, detail, and discipline. It's not uncommon for an experienced developer to design and develop the conceptual, logical, and physical model on the fly for a modest application database (say, a simple web application or the like). An organization investing in a large-scale system will (or should) employ a well-defined process, facilitated by database design experts, perhaps in concert with other analysis and design processes (such as Object-Oriented Analysis and Design (OOAD)).

One simple exercise for beginning the Conceptual Design is to start with a textual description of the application or database domain. In the text, start by picking out the nouns — these are potential entities and attributes. The verbs can indicate relationships. For example:

> The **company** *maintains* **warehouses** in several **locations** to *fulfill* **customer** needs. Each warehouse *has a* **warehouse identification number**, **name**, **facility description**, and **location identification number**.
>
> Each **customer** *has an* **identification number**. Customer records include **customer name**, **street name**, **city** or **state**, **country**, **phone numbers** (up to five phone numbers for each customer), and **postal code**. Some customers *place* **orders** through the Internet, so **e-mail addresses** are also recorded.

"Has-a" and "is-a-part-of" constructs can indicate entity attributes. Action verbs ("...customers *place* orders...") can indicate relationships between entities.

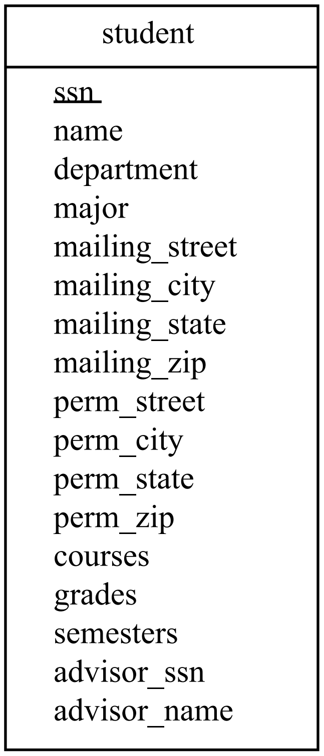
## Normalization

- *Normalization* defines a set of incremental tests to assure a database design will be free of redundancy and associated data anomalies.
    - A database (or subset of its tables) for which a certain level of normalization is true is said to be in that *normal form*.
    - Most database designers try to achieve at least third normal form (3NF) with their designs, and perhaps higher (or lower) levels for certain tables.
- To be in first normal form (1NF), several things must be true of a table:
    - Each row is unique.
    - For each row, each column contains at most one atomic value.
        - An atomic value cannot be decomposed into smaller values — for example, a **fullname** attribute is non-atomic, since it can be decomposed into **firstname** and **lastname** attributes.
    - For each row, each column value is non-repeating.
        - For example, a single **Customer** record cannot contain the list of that customer's **Order**s.
        - Redefine a repeating attribute as a separate entity.
- A table in 1NF has a defined primary key.
    - Like all other attributes, a primary key column is non-decomposable.
    - A table may have a composite primary key — in which the combination of values of the primary key columns uniquely identifies each row.

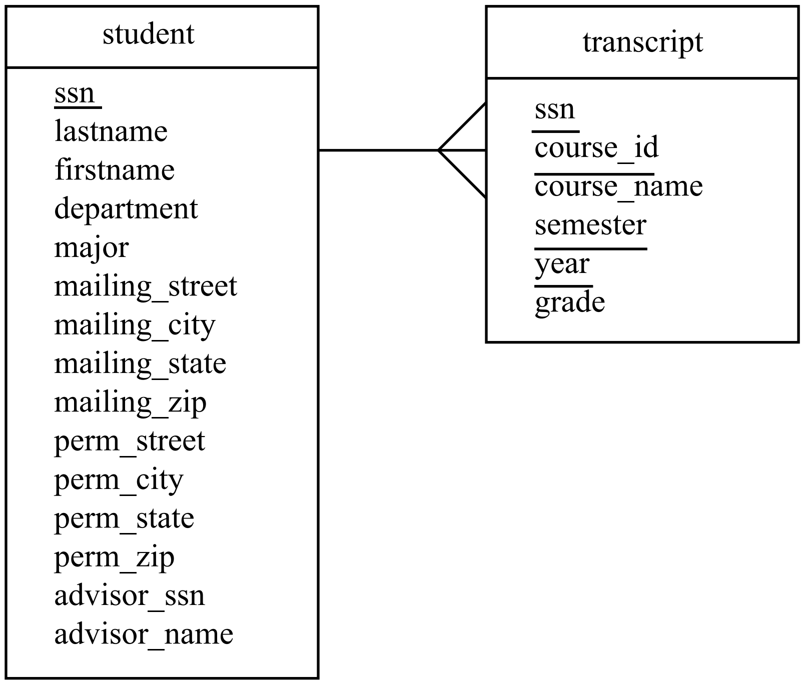
*Normalization* is a process of decomposing a table into one or more tables. There are multiple levels of normalization that can be applied. Each higher level of normalization adds more constraints onto the previous. Each normal form is defined by a set of rules that prohibit data redundancy. This prevents data inconsistencies on data updates, resulting in a more stable database.

The higher you go in normal forms, the more tables you will have to navigate to find your information resulting in slower performance. Though there are multiple levels of normalization a database can undergo, be careful not to compromise your performance needs by over-normalization.

non-1NF



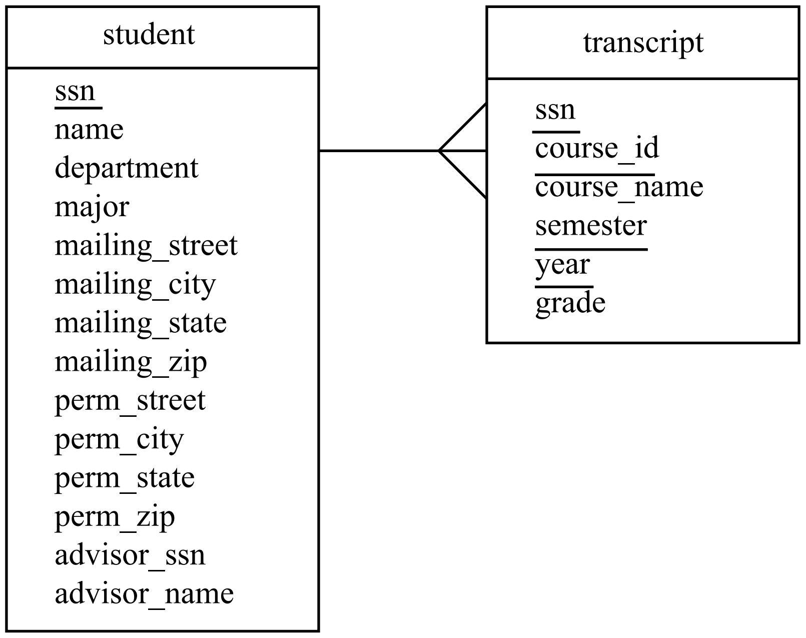
The **name**, **courses**, **grades**, and **semesters** attributes are not atomic. The **student** table is put into 1NF by splitting **name** and creating an additional table: **transcript**.



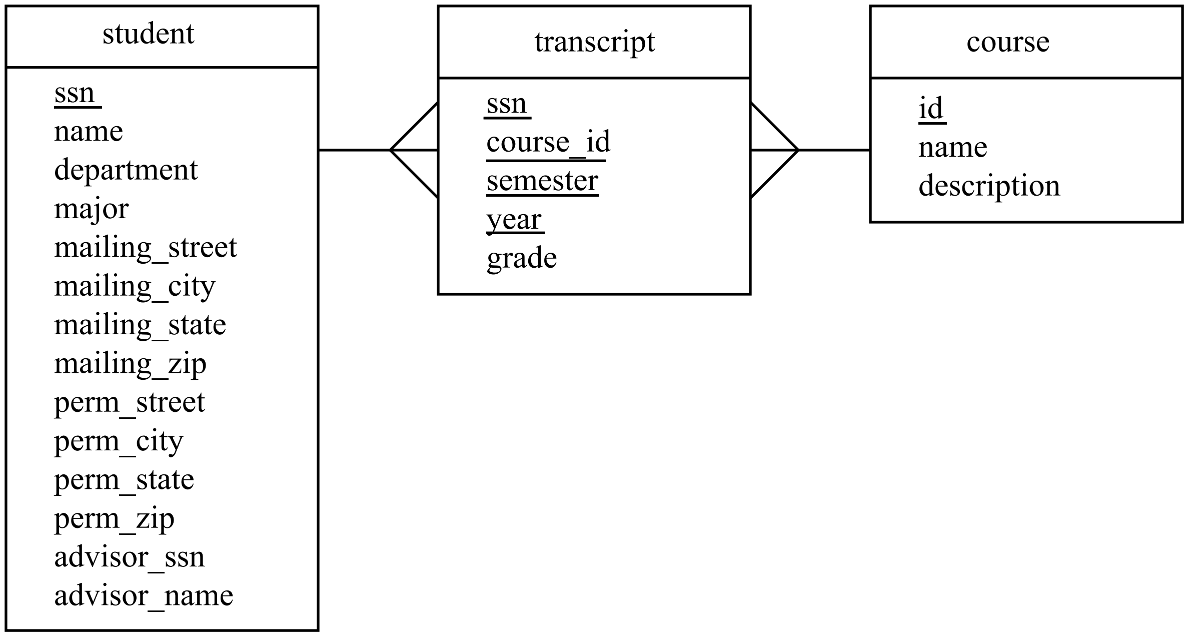
## Second and Third Normal Forms

- A table is in second normal form (2NF) if:
  - The table is in 1NF.
  - Each attribute depends on the entire primary key.
    - A 1NF table that does not have a composite primary key is already in 2NF.
- A table is in third normal form (3NF) if:
  - The table is in 2NF.
  - Each non-key attribute depends only on the primary key.
    - A 3NF table is a 2NF table with no non-key attribute dependent on another non-key column.

The **transcript** table has a composite primary key consisting of the following attributes: **ssn**, **course_id**, **semester**, **year**.



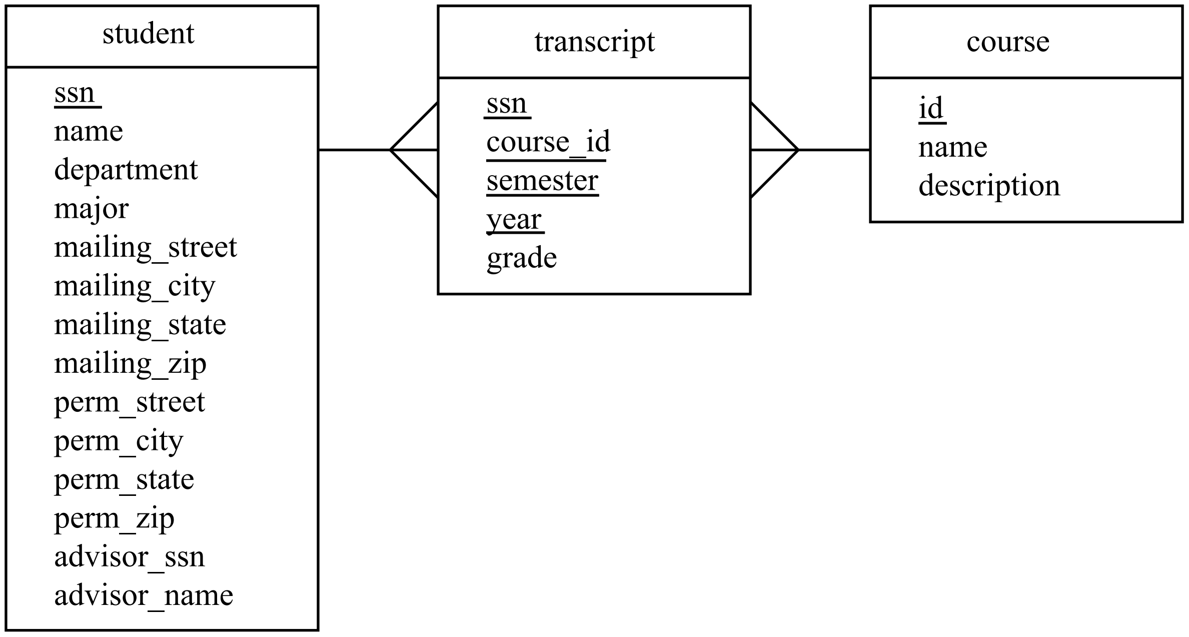
The **grade** is functionally dependent on the entire primary key. The **course_name** attribute depends only on the **course_id**. To meet 2NF, move **course_name** to a separate entity.



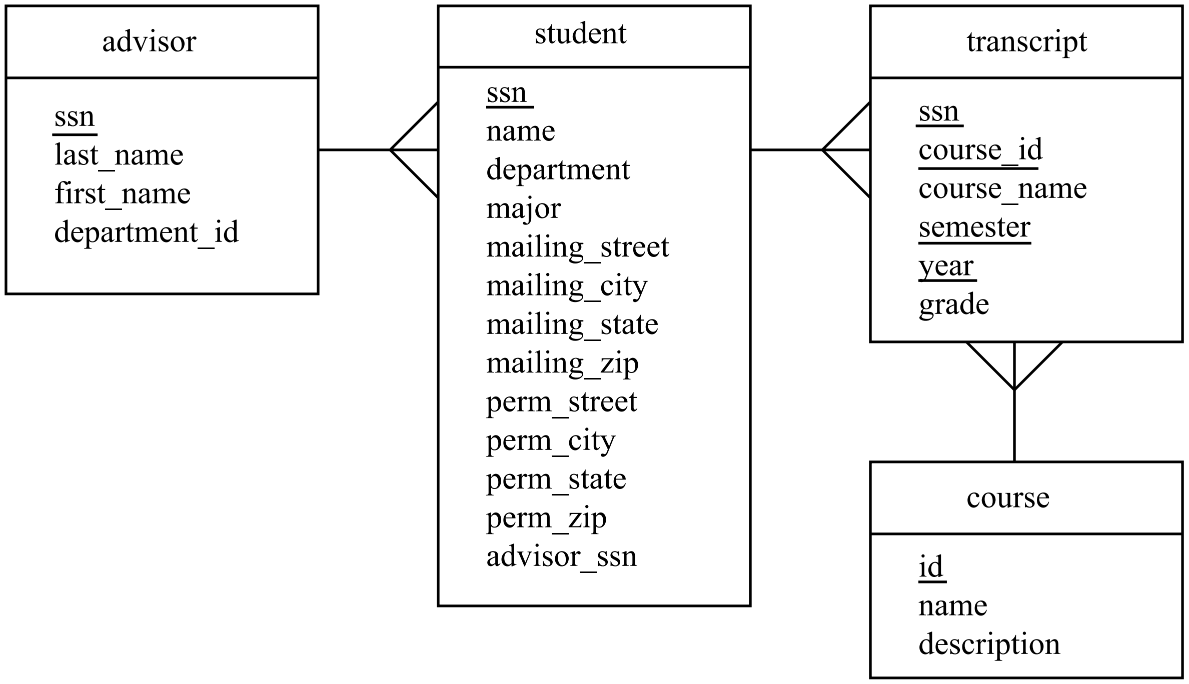
## OTHER NORMAL FORMS

- Additional normal forms, each building on the previous one, have been defined by relational database theorists.
  - Boyce-Codd Normal Form (BCNF).
  - Fourth Normal Form (4NF).
  - Fifth Normal Form (5NF).
  - Domain-Key Normal Form (DKNF).
  - Sixth Normal Form (6NF).
  - It is not necessarily desirable to apply these to every database.
    - Some are limited only to very particular types of tables.
    - Some are not always achievable.
- Normalization results in a proliferation of tables and elimination of redundant columns.
  - Sometimes this can lead to inconvenient complexity and poor performance.
- *Denormalization* is the deliberate and judicious violation of normalization rules, usually to provide convenient access and higher performance.
  - Denormalization results in redundant storage of an attribute in multiple tables.
  - This often occurs, for example, in data warehouses.

The **advisor_name** attribute in the student table is dependent on the non-key **advisor_ssn** attribute.



Place the **student** table in 3NF by moving the **advisor_name** attribute to another entity.



## APPLICATIONS FOR RELATIONAL DATABASES

- Perhaps the most familiar category of RDB is the *on-line transaction processing* (OLTP) database.
  - An OLTP database supports large volumes of concurrent data manipulation language (DML) access during normal operation.
  - Order processing, human resources, and web commerce systems are some typical OLTP applications.
  - OLTP databases are usually fully normalized, in order to maintain the "ACID" properties of transactions:
    - Atomicity
    - Consistency
    - Isolation
    - Durability
- A *data warehouse* is designed and optimized for queries and analysis rather than data manipulation.
  - Data warehouses frequently make use of denormalization and redundant copies of attribute values.
  - Data warehouse tables are often read-only, to prevent any possibility of update anomalies.
  - The data warehouse comprises the long-term historical record of database activity, usually derived from OLTP systems.
    - Part of a data warehouse design includes the procedures for *extraction, transformation, and loading* (ETL) of production data.
    - The data warehouse supports *on-line analytical processing* (OLAP).
- Other special-purpose applications include *spatial* and *temporal* databases.

Normalization prevents insert, update, and deletion anomalies in an OLTP database. Normally, a data warehouse is non-volatile — that is, non-updatable, once the data is loaded. This allows the data warehouse designer to violate a number of classic database design rules in order to improve performance or facilitate particular kinds of analysis.

For example, *Materialized Views* (formerly known as *snapshot tables*) physically replicate data that's present in existing fact tables in order to provide more convenient query capabilities.

The transformation portion of a data warehouse's ETL solution may include aggregation and derivation of values. For example, to facilitate weekly sales reporting, individual sale attributes may be aggregated together to weekly granularity, and stored. Or, counts of inventory items may be pre-calculated and stored. In an OLTP database, such storage of values which are readily derivable from other values (which may change over time) can result in inconsistencies and faulty analysis. In read-only data warehouse tables, this is not a problem.

Because data warehouses accumulated ever-growing amounts of historical data, and because they frequently store multiple copies of data values, they tend to have very high storage, memory, and analytical processing requirements.

## Labs

For this lab, we will describe requirements for two database applications: a Human Resources (HR) and an Order Entry (OE) system. For each, based on the description, we need to:

1. Identify the core entities and their important attributes.
2. Determine the cardinality (one-to-one, one-to-many, many-to-many) of the associations between entities.
3. Develop a logical model for the database:
   a. Determine table and column names.
   b. Determine primary key columns.
   c. Normalize, to 3NF, where possible.
   d. Identify foreign key references between tables.

Where possible, you can do this exercise in groups with other students. It is not necessary to produce a complete logical or physical model for both systems — just analyze and design the core tables.

(Solution: There is no single "correct" outcome for this exercise. After you are satisfied with your logical schema (or you run out of time), locate the Sample Schemas book in the Oracle Database online Documentation Library. Review the rationale, diagrams, and schema scripts for the Sample Schemas that Oracle provides. Compare them to your own design for further ideas on database analysis and design.)

HR:
Each employee is assigned an ID number. We need to record their name, email , job code, salary (some employees also earn commission), and manager ID.

Positions in the company are predefined, with a job code, job title, and a minimum and maximum salary range for the each. As employees advance (or otherwise change positions), we want to record the employment duration on that job, the job identification number, and the department.

The company has presence in several regions, and has several locations in which warehouses are located. There are many departments (Administration, Marketing, Purchasing, Human Resources, Shipping, etc.). Each employee is assigned to a department, and each department has a unique department ID. Each department is associated with one location, which has a full address that includes the street name, postal code, city, state or province, and the country code.

For each of the various locations associated with warehouses and departments, we record the country name, currency symbol, currency name, and the geographical region.

OE:
For our products, such as computer hardware and software, music, clothing, and tools, we store information such as product ID, product category, order entries, the shipping weight group, the warranty period, the supplier, the availability, a list price, a minimum retail price, and a URL for manufacturer information. Inventory information includes the warehouse where the product is available and the quantity on hand. Each warehouse has a warehouse ID number, name, facility description, and location ID.

Each customer has an ID number. Customer information includes customer name, street name, city or province, country, phone numbers (up to five phone numbers per customer), and zip code. Some customers place orders online, so we store their email addresses. Customers have a credit limit. Some customers have an Account Manager at our company.

When a customer places an order, we record the date of the order, how it was placed, the current status, shipping mode, total amount, and the Sales Representative who served the customer. The Sales Representative may or may not be the same person as the customer's Account Manager. If an order is placed online, no sales rep is involved. Of course the order information includes the number of items ordered, the unit price, and the products ordered.

# CHAPTER 12 - DATA WAREHOUSE FEATURES

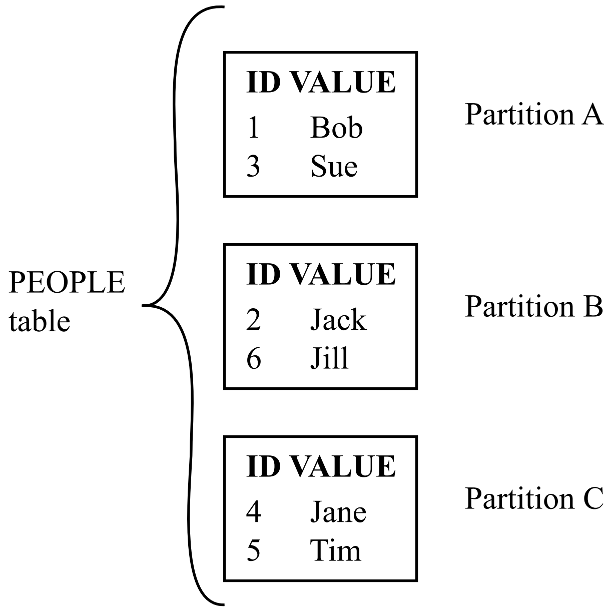
## OBJECTIVES

- ❅ Leverage partitioning for efficient queries over large amounts of data.
- ❅ Use parallel execution to speed up SQL results.
- ❅ Take advantage of Oracle's SQL features for both DML and query processing.

## PARTITIONED TABLES

- Partitioning lets you break down a table's storage into smaller, more manageable pieces, based on specified data values.
  - Each piece is called a *partition*.
  - A partition can be broken down further into *subpartitions*.
- A *partition key* is a set of one or more columns, the values of which determine which partition a row of data resides in.
- SQL queries do not have to be modified to deal with partitioned tables.
  - Partitioning can be totally transparent to the application.
  - Each partition can be queried independently if desired.
- Partitioning improves both performance and availability for large tables.
  - Partition pruning and partition-wise joins speed up query processing.
  - Older partitions can be placed in **READ ONLY** tablespaces, easing demands on backups.
  - Partitions are often placed in separate tablespaces, each residing on a different disk unit.
    - If you lose a disk unit, you only lose the data in that partition.
- Indexes can be partitioned, too.
- Each partition of a table or index must have the same logical structure.
  - Each partition of a table or index can have a different physical structure.

A table can easily be broken down into multiple partitions. In the diagram below, we can query the **PEOPLE** table, or query the individual partitions.



The query:

```
SELECT value FROM people;
```

will return **Bob**, **Sue**, **Jack**, **Jill**, **Jane**, and **Tim**.

The query:

```
SELECT value FROM people PARTITION (partition_b);
```

will return only **Jack** and **Jill**.

The **DBA_TAB_PARTITIONS** Data Dictionary view will show how a table is partitioned.

## PARTITIONING METHODS

- **RANGE** partitioning maps rows to partitions based on a range of values.
  - A common implementation is to partition based on a date field.
    - For example, each partition would hold a month, a quarter, or a year's worth of data.
  - Each partition has a **VALUES LESS THAN** clause, which defines the bounds of the range.
- **LIST** partitioning maps rows to partitions based on a defined list of values.
  - Use list partitioning if the partition key will only have a small number of distinct values.
- **HASH** partitioning uses a hash function on the partition key to map the row to a specific partition.
  - Hash partitioning is useful when the partition key does not lend itself to range or list partitioning.
- *Composite* partitioning starts with range partitioning and subpartitions by hash or list partitioning.
  - Composite range-hash partitioning is great for dividing the data by date and then subdividing into an even number of buckets.
  - Composite range-list partitioning is often used when dividing sales data by date and then by geographical region.

A range-partitioned table:

sales_data.sql

```
CREATE TABLE sales_data (
            sales_tx_id   NUMBER(6,0),
            sales_amount  NUMBER(6,2),
            sales_date    DATE
) PARTITION BY RANGE (sales_date)
  (PARTITION sales_q1 VALUES LESS THAN
       (TO_DATE('04/01/2006','MM/DD/YYYY')),
   PARTITION sales_q2 VALUES LESS THAN
       (TO_DATE('07/01/2006','MM/DD/YYYY')),
   PARTITION sales_q3 VALUES LESS THAN
       (TO_DATE('10/01/2006','MM/DD/YYYY')),
   PARTITION sales_q4 VALUES LESS THAN
       (TO_DATE('01/01/2007','MM/DD/YYYY')));
```

The **SALES_DATE** column is the partition key.

A row with **SALES_DATE** equal to **12/13/2006** will be stored in partition **SALES_Q4**. A row with **SALES_DATE** equal to **01/26/2006** will be stored in partition **SALES_Q1**. There is no lower bound in range partitioning. If you need to implement a lower bound, then you will have to use a constraint to ensure the lower bound is enforced. If you desire no upper bound, then use the **VALUES LESS THAN MAXVALUE** clause when creating the partition. A list-partitioned table:

students.sql

```
CREATE TABLE students (
            student_id   NUMBER,
            student_name VARCHAR2(60),
            student_year CHAR(2)
              CHECK (student_year IN ('FR','SO','JR','SR')))
PARTITION BY LIST (student_year)
  (PARTITION freshmen VALUES ('FR'),
   PARTITION sophomores VALUES ('SO'),
   PARTITION juniors VALUES ('JR'),
   PARTITION seniors VALUES ('SR'));
```

A **PARTITION VALUES (DEFAULT)** clause will catch all partition key values not explicitly listed in any other partition. A hash-partitioned table:

employees.sql

```
CREATE TABLE employees (
            ssn   NUMBER,
            name  VARCHAR2(50))
PARTITION BY HASH (ssn)
PARTITIONS 5
STORE IN (ts1, ts2, ts3, ts4, ts5);
```

## Partition Pruning and Partition-wise Joins

- ❈ Partition pruning eliminates unneeded partitions from consideration when processing a SQL statement.
  - ➢ Accessing only those partitions that will participate in the query can greatly improve performance.
  - ➢ The Optimizer will automatically perform partition pruning when possible.
  - ➢ The partition key should be chosen carefully so that most common queries on the table can be supported by partition pruning.
- ❈ A partition-wise join occurs when two tables are joined on the partition key columns.
  - ➢ A partition-wise join can be a great performance boost to processing the join operation.

In a range-partitioned table **ACCOUNTS_PAYABLE**, each partition is one quarter's worth of accounts payable information.

Q1

| ID | A_DATE | VALUE |
|---|---|---|
| 1 | 01/26/06 | 101.34 |
| 2 | 03/10/06 | 99.23 |

Q3

| ID | A_DATE | VALUE |
|---|---|---|
| 10 | 07/01/06 | 183.34 |
| 15 | 08/08/06 | 127.79 |

Q2

| ID | A_DATE | VALUE |
|---|---|---|
| 7 | 04/12/06 | 87.33 |
| 8 | 04/18/06 | 91.23 |

Q4

| ID | A_DATE | VALUE |
|---|---|---|
| 21 | 10/31/06 | 10.23 |
| 25 | 11/28/06 | 88.45 |

A user issues the following query, looking for August's accounts payable information:

```
SELECT  a_date,value
  FROM  accounts_payable
 WHERE  a_date BETWEEN TO_DATE('08/01/06', 'MM/DD/YY')
   AND  TO_DATE('08/31/06', 'MM/DD/YY');
```

The Optimizer will automatically prune all unnecessary partitions. In this example, only partition Q3 will participate in the query.

Similar to the **ACCOUNTS_PAYABLE** table, we also have a **RECEIPTS** table. The **RECEIPTS** table is also partitioned by the fiscal quarter. Someone in accounting is looking to summarize the accounts payable by receipt and issues the following query:

```
SELECT  r.receipt_id, ap.id, ap.value
  FROM  accounts_payable ap JOIN receipts r
                              ON (r.ap_id = ap.id)
 WHERE  ap.a_date BETWEEN TO_DATE('01/01/06', 'MM/DD/YY')
   AND  TO_DATE('06/30/06', 'MM/DD/YY');
```

Partition pruning will eliminate all but the first and second quarter. Since the **ACCOUNTS_PAYABLE** and **RECEIPTS** tables are both partitioned on the join column, the Optimizer will also perform a partition-wise join on the remaining partitions.

## BITMAP INDEXES

- Use bitmap indexes on columns with a relatively low cardinality — those with only a few discrete values.
  - Creating a bitmap index is nearly identical to creating a regular, B-tree index.

```
CREATE BITMAP INDEX sales_region_bidx
                ON sales(region);
```

  - The Optimizer will ignore B-tree indexes on columns with low cardinality.
- A bitmap will be created for each possible value in a column.
  - Each bit in the bitmap will point to a specific row in the table.
  - If the column contains the value in that row, the bit for that row will be turned on; else the bit will be off.
- Bitmaps typically offer performance benefits for data warehouse applications.
  - Bitmap indexes often require less space than B-tree indexes.
  - Mutiple bitmaps can be logically **AND**'d or **OR**'d together to help query performance.
    - Bitmap indexes are most effective when used with multiple conditions in the **WHERE** clause.
- Bitmap indexes are not well suited for OLTP applications, which typically experience a high degree of concurrent DML operations.

In this **SALES** table, we have three regions: **East**, **West**, and **Central**. The low number of distinct values lends itself well to a bitmap index. We can see the bitmaps next to the rows of data in the following diagram:

| SALES_TABLE | | |
|---|---|---|
| **ID** | **S_DATE** | **REGION** |
| 1 | 01/01/06 | East |
| 3 | 03/12/06 | West |
| 4 | 01/02/06 | Central |
| 7 | 06/05/06 | West |
| 8 | 03/12/06 | Central |

| SALES_REGION_BIDX | | |
|---|---|---|
| **East Bitmap** | **Central Bitmap** | **West Bitmap** |
| | | |
| 1 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |

The bitmaps can be logically combined depending on the **WHERE** clause in your query. Assume the user issues the following query:

```
SELECT * FROM sales
 WHERE region = 'East' OR region = 'West';
```

The bitmaps for the east and west regions can be logically **OR**'d together as follows:

| SALES_REGION_BIDX | | |
|---|---|---|
| **East Bitmap** | **Central Bitmap** | **East OR West** |
| | | |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

The resulting bitmap of the **OR** operation now points to all rows where the region is **East** or **West**.

## MATERIALIZED VIEWS

- A *materialized view* stores the results of a pre-defined query.
    - Typically, the materialized view's base query performs summary or other aggregation operations.
- Materialized views are used to improve performance of long-running queries or to replicate data.
- If possible, the Optimizer will rewrite a query to use the materialized view instead of the base tables.
    - This *Query Rewrite*, using a materialized view, is transparent to the user.
    - A user can query a materialized view directly if they choose.
- The SQL Access Advisor in Oracle10*g* can be used to determine which queries would benefit from a materialized view.

In your data warehouse, you might have 3 tables containing sales data. These tables are frequently joined together with a query similar to the following:

```
   SELECT region_name, AVG(sales_value)
     FROM salesman sm, sales_data sd, sales_region sr
    WHERE sm.region_id =  sr.region_id
      AND sm.salesman_id = sd.salesman_id
 GROUP BY region_name;
```

The DBA creates a materialized view with the following statement:

```
CREATE MATERIALIZED VIEW sales_region_mv
  TABLESPACE users
       BUILD IMMEDIATE
     REFRESH COMPLETE ON DEMAND
      ENABLE QUERY REWRITE AS
      SELECT region_name, AVG(sales_value), SUM(sales_value),
             MAX(sales_value), MIN(sales_value)
        FROM salesman sm, sales_data sd, sales_region sr
       WHERE sm.region_id =  sr.region_id
         AND sm.salesman_id = sd.salesman_id
    GROUP BY region_name;
```

If a user issues the first query the materialized view will be used instead, saving a great deal of processing time. The materialized view will even be used for this query:

```
  SELECT region_name, MIN(sales_value), MAX(sales_value)
    FROM salesman sm, sales_data sd, sales_region sr
   WHERE sm.region_id =  sr.region_id
     AND sm.salesman_id = sd.salesman_id
GROUP  BY  region_name;
```

## CREATING MATERIALIZED VIEWS

- You must have been granted the **CREATE MATERIALIZED VIEW** system privilege, and either the **CREATE TABLE** or **CREATE ANY TABLE** system privilege.
- The **BUILD** clause determines if the materialized view is immediately populated with data, or populated later.
    - The default is **IMMEDIATE**.
- You must enable query rewrite if you want the materialized view to be eligible for use by other queries.
    - The default is **DISABLE QUERY REWRITE**.

```
CREATE MATERIALIZED VIEW mview_name
       [TABLESPACE ts_name]
       [<storage_clause>]
       [BUILD IMMEDIATE|DEFERRED]
       [REFRESH FAST|COMPLETE|FORCE]
       [ENABLE|DISABLE QUERY REWRITE]
    AS <subquery>;
```

Disabling query rewrite for a materialized view is still useful. You may not want your users to enjoy query rewrite for their queries if the contents of the materialized view is stale. However, you may want aggregate queries that do not require the latest and greatest data to be able to use the contents of the materialized view. In this case, you disable query rewrite for the materialized view and query the materialized view directly. You can change the query rewrite ability of a materialized view with the **ALTER MATERIALIZED VIEW** command.

Query rewrite has the following restrictions:

- If a column appears in the base query's **GROUP BY** clause, it must also appear in the columns of the **SELECT** clause.
- You cannot have a **CONNECT BY** clause in the query.
- The base tables and the materialized view cannot be owned by the **SYS** user.
- If the materialized view base query has both local and remote tables, only queries on the local tables are eligible for query rewrite.

## REFRESHING MATERIALIZED VIEWS

- There are three ways to refresh the materialized view's contents from the base tables.

    - A **FAST** refresh will be updated as changes occur to the base tables, but it requires the most overhead.

    - A **COMPLETE** refresh updates the materialized view by rerunning the base query.

    - Specifying **FORCE** indicates to perform a **FAST** refresh if possible, else perform a **COMPLETE** refresh.

        - **FORCE** is the default.

Before you can create a fast refresh materialized view, you must create materialized view logs on each base table. Writing to the materialized view logs for each transaction on the base table incurs overhead, so weigh the pros and cons of fast refresh carefully. Creating a materialized view log is as simple as the following command:

```
CREATE MATERIALIZED VIEW LOG ON schema.table_name;
```

If using fast refresh, you have an optional parameter: **ON COMMIT** or **ON DEMAND**. The **ON COMMIT** parameter indicates that the fast refresh is to take place on every commit of the base tables. The **ON DEMAND** parameter indicates that you will perform the fast refresh by calling the **DBMS_MVIEW** supplied package. Fast refresh with **ON COMMIT** requires the most overhead, so it should be used carefully.

Not all materialized views can be fast refreshed. The **DBMS_MVIEW.EXPLAIN_MVIEW** procedure can tell you if the materialized view can be fast refreshed.

## The MERGE Statement

- The **MERGE** statement lets you select rows of data from a source table and either insert new rows into a destination table, or update existing rows in that destination table.

```
MERGE INTO dest_table
           USING source_table
        ON (condition)
      WHEN MATCHED THEN UPDATE . . .
      WHEN NOT MATCHED THEN INSERT . . .;
```

  - Before the **MERGE** statement was available, you had to perform separate **INSERT** and **UPDATE** statements.

- The **INTO** clause denotes the destination table and the **USING** clause indicates the source table.

- With the **ON** clause you can define a test condition to be applied to each source row.

- The **WHEN MATCHED** clause specifies how to update existing rows for those source rows that match the test condition.

- For rows that do pass the test condition, the **WHEN NOT MATCHED** clause specifies how to insert the new rows into the destination table.

You have been given a table of employee information. In this table, some employees are not yet added to the **HR.EMPLOYEES** table. For those employees that are found in the table, we need to ensure that the email addresses and phone numbers are up-to-date. An example of a **MERGE** statement which can accomplish this task in a single SQL statement can be seen below:

```
MERGE INTO employees dest
            USING emp_source_list source
         ON (source.emp_id = dest.employee_id)
WHEN MATCHED THEN
   UPDATE SET dest.email = source.email_address,
              dest.phone_number = source.phone
WHEN NOT MATCHED THEN
   INSERT (dest.employee_id,dest.first_name,dest.last_name,
           dest.email,dest.phone_number,dest.hire_date)
   VALUES (source.emp_id,source.first,source.last,
           source.email_address,source.phone,SYSDATE);
```

## Multi-table INSERT Statements

❈ Like the name implies, the *multi-table **INSERT** statement* will add data to one or more tables with a single SQL command.

- ➢ A multi-table **INSERT** statement is more efficient than multiple single-table **INSERT** statements.

❈ A multi-table **INSERT** statement can contain conditional logic to denote which tables will receive the rows of data.

```
INSERT [ALL|FIRST]
   [WHEN expr THEN] INTO table1 VALUES ...
   [WHEN expr THEN] INTO table2 VALUES ...
   [ELSE INTO tableN VALUES ...]
SELECT ...;
```

❈ The optional **WHEN** clauses will determine if a table receives the row of data.

- ➢ You can have at most 127 **WHEN** clauses.
- ➢ The optional **ELSE** clause is a catch-all condition if all other **WHEN** conditions fail.

❈ If using **WHEN** clauses, you can have the row insert into **ALL** tables that have a true condition or just the **FIRST** true condition.

- ➢ If you are not using **WHEN** clauses, then you cannot use the **FIRST** clause.

A common use of a mutli-table **INSERT** statement is to insert data into multiple tables at the same time, similar to the following:

person.sql
```
INSERT ALL
   INTO person (first, last) VALUES (first_name, last_name)
   INTO subject (full_name)
         VALUES (first_name || ' ' || last_name)
SELECT employee_id, first_name, last_name
  FROM hr.employees;
```

A nice feature of the multi-table **INSERT** statement is to insert slightly modified data, multiple times, into the same table.

- Add sales forecasts for the next four quarters.
- Each quarter will be have a 10% increase in the forecast.

```
INSERT ALL
   INTO forecast VALUES (sysdate,region, avg_sales*1.10)
   INTO forecast VALUES (sysdate+90, avg_sales*1.21)
   INTO forecast VALUES (sysdate+180, avg_sales*1.33)
   INTO forecast VALUES (sysdate+270, avg_sales*1.46)
SELECT region, AVG(sales_value) AS avg_sales
  FROM sales GROUP BY region;
```

An example of a conditional multi-table **INSERT**:

```
INSERT FIRST
   WHEN avg_sales < 1000 THEN
       INTO poor_regions VALUES (region, avg_sales)
   WHEN avg_sales < 100 THEN
      INTO terrible_regions VALUES (region)
   WHEN avg_sales < 9999 THEN
       INTO good_regions VALUES (region, avg_sales)
   ELSE
       INTO great_regions VALUES (region, 'Doing Excellent!')
SELECT region, AVG(sales_value) AS avg_sales
  FROM sales GROUP BY region;
```

Note in the example above, no data will ever be inserted into the **TERRIBLE_REGIONS** table due to the **FIRST** clause.

## PARALLEL STATEMENTS

- A SQL statement can be broken down into smaller units, each of which is run in parallel (at the same time).
    - All of the results from each unit of work are combined into one result set.
- For demanding SQL statements, parallel execution results in a shorter processing time.
    - Short-running SQL statements will most likely not benefit from parallelism.
- A *query coordinator* oversees the parallel execution.
    - The query coordinator will acquire multiple slave processes.
    - Each slave process performs the work in parallel.
    - The query coordinator is responsible for combining the output from the slave processes into the final result.
- The degree of parallelism is defined as the number of slave processes performing the work for a single operation.
    - The degree or parallelism should be no more than twice the total number of CPUs on your database server.

Parallel queries are useful in the following situations:

- Queries requiring large table scans
- Queries involving join operations
- Queries on partitioned tables
- Creating large indexes or materialized views

Parallel queries are not necessarily useful under any of the following circumstances:

- Your database server only has one CPU
- The CPU units on your database server typically experience high usage
- There is not enough free physical memory on the server
- Your database server does not have sufficient I/O bandwidth

The Oracle database contains parameters to help control parallel execution. The database will pre-start a number of slave processes denoted by the **PARALLEL_MIN_SERVERS** initialization parameter. The **PARALLEL_MAX_SERVERS** initialization parameter defines the maximum number of slave processes available to all sessions. The default values for both parameters are normally useful for most Oracle instances.

The degree of parallelism is defined for the SQL statement by the following, in order:

1. The degree noted in the **PARALLEL** hint in the SQL statement.

2. An **ALTER SESSION FORCE QUERY PARALLEL** command.

3. The default degree of parallelism associated with the table, either with the **CREATE TABLE** or **ALTER TABLE** commands.

4. The default degree of parallelism associated with an index used in the execution plan.

A parallel query, with degree of parallelism set to 4:

```
SELECT /*+ PARALLEL(s 4) PARALLEL(t 4) */
       r.region,  MAX(s.sales_value)
  FROM sales s, regions r
 WHERE s.region_id = r.region_id;
```

DML operations can be performed in parallel:

```
UPDATE /*+PARALLEL(employees 8) */ SET salary = salary*1.1;
```

## Labs

❶ Create a table, **EMP_REGIONS**, with four columns: **EMPLOYEE_ID**, **FIRST_NAME**, **LAST_NAME**, and **REGION_NAME**. The column types should match the corresponding columns from **HR.EMPLOYEES** and **HR.REGIONS**. Your table will be list-partitioned, with four partitions — one for each region. Populate the **EMP_REGIONS** table with data from **HR.EMPLOYEES** and **HR.REGIONS** table. (Hint: You will need the **HR.DEPARTMENTS**, **HR.LOCATIONS**, and **HR.COUNTRIES** tables in your join.)
(Solution: *cr_regions.sql*)

❷ Query the **EMP_REGIONS** table to show the data and also query each partition individually to see the data has been populated into the correct partition.
(Solution: *regions_query.sql*)

❸ In SQL*Plus, set the **TIMING ON** option. Run this query on the **SH** schema's **SALES** table:

```
  SELECT promo_id, MAX(amount_sold)
    FROM sh.sales
GROUP BY promo_id
```

Repeat the query a few times, noting the execution time. Now, create a normal view, **V_PROMO_SALES**, using this query. Run a **SELECT * FROM v_promo_sales** query a few times, again noting the execution time. Finally, create a materialized view, **MV_PROMO_SALES**, using the same query. Take note of the time necessary to execute the **CREATE MATERIALIZED VIEW** statement. Run **SELECT * FROM mv_promo_sales** a few times, noting the execution time.
(Solution: *cr_promo_views.sql*)

❹ In SQL*Plus, set **AUTOTRACE ON EXPLAIN**. Alter the **MV_PROMO_SALES** materialized view, to **ENABLE QUERY REWRITE**. Now, do an **ALTER SESSION** to set **QUERY_REWRITE_ENABLED** to **FALSE**. Execute the query above and note the execution plan. Finally, do another **ALTER SESSION** to set **QUERY_REWRITE_ENABLED** to **FORCE**. Reexecute the query, and note the execution plan, as well as the execution time. Turn **AUTOTRACE** off when you finish.
(Solution: *query_rewrite.sql*)

❺ Do the following with a single **INSERT** statement:

- Add every employee whose first or last name begins with 'A' to the 'Asia' region in **EMP_REGIONS**.
- Add every employee whose first or last name begins with 'M' to the 'Middle East' region in **EMP_REGIONS**.

(Solution: *insert_first.sql*)